5.2.6. Executie neordonata si redenumirea registrelor

Numeroase UCP-uri moderne sunt atât în banda de asamblare cât si superscalare, asa cum se arata în Fig. 5.27. Ceea ce înseamna în general, ca exista o unitate de citire care citeste instructiuni din memorie înainte ca acestea sa fie necesare pentru a fi trimise unitatii de decodificare. Unitate de decodificare trimite instructiunile decodificate unitatilor functionale adecvate pentru executie. În anumite cazuri, instructiuni individuale se pot sparge în micro-operatii (micro-ops) înainte de a le trimite unitatilor functionale depinzând de ce sunt capabile sa faca unitatile functionale.

Evident, proiectarea masinilor este mai simpla daca toate instructiunile sunt executate în ordinea în care sunt citite (presupunând pentru moment ca algoritmul de predictie a ramiflcatiior nu greseste niciodata). Totusi, executia în ordine nu furnizeaza întotdeauna performanta optima din cauza dependentei între instructiuni. Daca o instructiune necesita o valoare calculata de catre instructiunea precedenta, cea de-a doua instructiune nu îsi poate începe executia pâna ce prima nu a produs valoarea necesara. În aceasta situatie (o dependenta RAW) a doua instructiune trebuie sa astepte. Exista de asemenea si alte tipuri de dependenta, asa cum vom vedea în curând.

În încercarea de a aborda aceste probleme si de a realiza o performamta superioara, anumite UCP-uri permit sarirea unor instructiuni pentru a ajunge la instrutiuni urmatoare care nu sunt dependente. Nu mai trebuie sa spunem ca algoritmul intern de planificare a instructiunilor utilizat trebuie sa furnizeze acelasi efect ca si când programul ar fi executat în ordinea scrisa . Vom demonstra acum cun functioneaza reordonarea instructiunilor utilizând un exemplu detaliat.

Pentru a ilustra natura problemei vom începe cu o masina care întotdeauna furnizeaza instructiuni în ordinea programului si de asemenea necesita terminarea executiei în ordinea programului. Semnificatia ideii va deveni clara mai târziu.

Masina din exemplul nostru are opt registre vizibile programatorului, de la R0 la R7. Toate instructiunilor aritmetice utilizeaza trei registre: doua pentru operanzi si unul pentru rezultat Vom presupune ca daca o instructiune este decodificata în ciclul n, executia începe în ciclul n+1. Pentru o instructiune simpla, cum ar fi o adunare sau o înmultire, scrierea în registrul destmatie se produce la sfârsitul ciclului n+2. Pentru o instructiune mai complicata, cum ar fi o înmultire, scrierea se produce la sfârsitul ciclului n+3. Pentru a apropia exemplul de realitate, vom permite unitatii de decodificare sa lanseze în executie pâna la doua instructiuni pe ciclu de ceas. UCP-urile superscalare comerciale adesea pot lansa în executie patru sau chiar sase instructiuni pe ciclu de ceas.

Secventa de executie din exemplul nostru este aratata în Fig. 5.31. Aici prima coloana furnizeaza numarul de cicluri si a doua coloana furnizeaza numarul instructiunii. A treia coloana listeaza instructiunea decodificata, a patra spune care instructiune este lansata în executie (cu un maxim de doua pe ciclu de ceas). A cincea spune care instructiune a fost retrasa (terminat). Sa ne amintim ca în acest exemplu cerem atât lansarea în executie în ordine cât si terminarea în ordine, astfel ca instructiunea k+1 nu poate fi lansata în executie pâna când instructiunea k nu a lost lansata în executie iar instructiunea k+1 nu poate fi retrasa (întelegând efectuarea scrierii în registrul destinatie) pâna ce instructiunea k nu a fost retrasa. Celelalte 16 coloane sunt discutate mai jos.

					Registre care sunt citite								Registre care sunt scrise							
C	#	Decodificat	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1	1		
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	-		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1*R2	5			2	1		1										1	1
6	6	R1=R0-R2				2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1							1						
	7	R3=R3*R1	7		1	1	1	1						1		1				
10					1	1	1	1						1		1				
11				6		1		1												
12				7																
13		R1=R4+R4	8						2					1						
14									2					1						
15				8																

C = Ciclu; Iss = Lansata; Ret = Retrasa

Figura 5.31. Functionarea unei UCP superscalar cu lansare în executie în ordine si terminare în ordine.

Dupa decodificarea unei instructiuni unitatea de decodificare trebuie sa decida daca poate fi lansata în executie imediat sau nu. Pentru a lua aceasta decizie, unitatea de decodificare trebuie sa cunoasca starea tuturor registrelor. Daca, de exemplu, instructiunea curenta necesita un registru a carui valoare nu a fost înca calculata, instructiunea curenta nu poate fi lansata în executie si UCP trebuie sa se blocheze.

Vom pastra informatii despre utilizarea registrelor folosind un dispozitiv numit tabela de scor (scoreboard), care a fost prima data prezent în CDC 6600. Tabela de scor are un mic numarator pentru fiecare registru indicând de câte ori este utilizat acel registru ca sursa de catre instructiunile aflate în executie. Daca maximum, sa zicem, 15 instructiuni se pot executa simultan atunci este necesar un contor de 4 biti. Când este lansata în executie o instructiune, intrarile tabelei de scor pentru registrele sale cu operanzi sunt incrementate. Când o instructiune este retrasa, intrarile sunt decrementate.

Tabela de scor are de asemenea un numarator pentru fiecare registru, pentru a pastra informatii despre registrele utilizate ca destinatii. Deoarece se permite o singura scriere la un moment dat, aceste numaratoare pot fi de 1 bit. În Fig. 5.31 cele 16 coloane din dreapta arata tabela de scot.

În masinile reale, tabela de scor pastreaza si informatii despre utilizarea unitatilor functionale pentru a împiedica lansarea în executie unei instructiuni pentru care nu este disponibila nici o unitate functionala. Pentru simplitate, vom presupune ca întotdeauna exista o unitate functionala adecvata disponibila, asa ca nu vom arata unitatile functionale pe tabeta de scor.

Prima linie din Fig. 5.31 arata II (instructiunea 1), care înmulteste R0 cu R1 si pune rezultatul în R3. Deoarece nici unul din aceste registre nu este deocamdata folosit, instructiunea este lansata în executie si tabela de scor actualizata pentru a reflecta faptul ca R0 si R1 sunt citite si R3 este înscris. Nici una din instructiunile urmatoare nu poate scrie în oricare din aceste registre sau nu poate citi R3 pâna când I1 nu a fost retrasa . Deoarece aceasta instructiune este o înmultire, ea se va încheia la sfârsitul ciclului 4. Valorile tabelei de scor aratate pe fiecare linie reflecta starea acestora dupa ce instructiunea de pe acea linie a fost lansata în executie. Intrarile vide (nemarcate) sunt zerouri.

Deoarece exemplul nostru este o masina superscalara care poate lansa în executie doua instructiuni pe ciclu, o a doua instructiune (I2) este lansata în executie în timpul ciclului 1. Aceasta aduna R0 si R2, memorând rezultatul în R4. Pentru a vedea daca aceasta instructiune poate fi lansata în executie se aplica urmatoarele reguli:

1. Daca orice operand este în curs de înscriere, nu se va lansa în executie (dependenta RAW).
2. Daca registrul rezultat este în curs de citire, nu se va lansa în executie (dependenta WAR).
3. Daca registrul rezultat este în curs de scriere, nu se va lansa în executie (dependenla WAW).

Am discutat deja dependentele RAW, care au loc când o instructiune trebuie sa utilizeze ca sursa un rezultat pe care o instructiune precedenta înca nu l-a produs. Celelalte doua dependente sunt mai putin serioase.

Ele reprezinta de fapt conflicte de resurse. Într-o dependenta WAR (Write After Read, rom: scriere dupa citire), o instructiune încearca sa supraînscrie un registru pe care o instructiune precedenta poate nu a tenininat înca sa-l citeasca. O dependenta WAW (Write After Write, TOJ scriere dupa scriere) este similara . Acestea pot fi adesea evitate pnin plasarea rezultatelor celei de-a doua instructiuni undeva în alta parte (eventual temporar). Daca nici una din cele trei dependente de mai sus nu exista si daca unitatea functionala necesana este disponibila, instructiunea este lansata în executie. În acest caz, I2 utilizeaza un registru (R0) care este în curs de citire de catre o instructiune în desfasurare, dar aceasta suprapunere este permisa deci I2 este lansata în executie. Asemanator, I3 este furnizata în timpul ciclului 2. Trecem acum la I4, care necesita utilizarea lui R4. Din nefericire, vedem din linia 3 ca R4 este în curs de scriere. Aici avem a dependenta RAW, asa ca unitatea de decodificare se blocheaza pâna când R4 devine disponibil. Cât timp este blocata, aceasta nu va prelua instructiuni de la unitatea de citire a instructiunilor. Când tamp oanele interne ale unitatii de citire se încarca integral, se va opri citirea instructiunii în avans.

Merita sa observam ca urmatoarea instructiune în ordinea programului, I5 nu are conflicte cu nici una din instructiunile în desfasurare. Ar fi putut fi decodificata si lansata în executie, dar nu este caci acest proiect necesita lansarea instructiunilor în executie în ordine.

Sa urmarim acum ce se întâmpla în timpul ciclului 3.I2, find o adunare (doua cicluri), se încheie la sfâsitul ciclului 3. Din nefericire, ea nu poate fi retrasa (eliberând astfel R4 pentru I4). De ce nu? Motivut este ca acest proiect necesita, de asemenea, retragerea în ordine. De ce? Ce prejudiciu posibil ar putea proveni din memorarea în R4 acum si marcarea acestuia ca disponibil?

Raspunsul este subtil, dar important. Sa presupunem ca instructiunile se pot termina în ne-ordine.

Atunci, daca apare o intrerupere, ar fi foarte dificil sa se salveze starea masinii, astfel încât sa se restabiteasca ulterior. În particular nu ar fi posibil sa se spuna ca toate instructiunile pâna la o anumita adresa au fost executate si toate instructiunile de dupa aceasta nu. Aceasta este numita intrerupere precisa (precise interrupt) si este o caracteristica dezirabila într-o UCP (Moudgill si Vassiliadis, 1996). Retragerea în ne-ordine face întreruperile imprecise, motiv pentru care unele masini necesita terminarea în ordine a instructiunilor.

Revenind la exemplul nostru la sfârsitul ciclului 4 toate cele trei instructiuni în desfasurare pot fi retrase, astfel ca în ciclul 5 I4 poate fi în sfârsit lansata în executie, alaturi de nou decodificata I5. Ori de câte ori o instructiune este retrasa unitatea de decodificare trebuie sa verifice daca exista o instructiune blocata care poate fi acum lansata în executie.

În ciclul 6, I6 se blocheaza pentru ca trebuie sa scrie în R1, iar R1 este ocupat. Ea este lansata în sfarsit în ciclul 9. Întreaga secventa de opt instructiuni consuma pentru terminare 15 cicluri datorita numeroaselor dependente, chiar daca hardware-ul este capabil sa lanseze în executie doua instructiuni în fiecare ciclu. Remarcati totusi, citind de sus în jos coloana Lansata din Fig. 5.31, ca toate instructiunile au fost lansate în executie în ordine. În acelasi fel, coboana Retrasa arata ca ele au fost retrase de asemenea în ordine.

Sa consideram acum o solutie alternativa: executia în ne-ordine. În aceasta solutie instructiunile pot fi lansate în executie în alta ordine si pot fi retrase de asemenea în alta ordine. Aceeasi secventa de opt instructiuni este aratata în Fig. 5.32, numai ca acum sunt permise lansarea în executie si retragerea în ne-ordine.

Prima diferenta se produce în ciclul 3. Chiar daca I4 s-a blocat, putem decodifica si lansa I5 pentru ca nu este în conflict cu nici o instructiune în desfasurare. Sarirea peste instructiuni genereaza totusi o noua problema. Sa presupunem ca I5 a utilizat un operand calculat de instructiunea peste care s-a sarit, I4. Cu tabela curenta de scor, nu am remarcat acest lucru. Ca o consecinta trebuie sa extindem tabela de scor pentru a retine informatii despre memorarile facute de instructiunile peste care s-a sarit. Aceasta se poate face adaugând un al doilea plan de biti, un bit pentru fiecare registru, pentru a retine informatii despre memorarile facute de instructiunile blocate (aceste contoare nu sunt aratate în figura). Acum regula pentru lansarea instructiunilor în executie trebuie extinsa pentru a evita lansarea oricarei instrutiuni cu un operand planificat sa fie modificat de o instructiune care a sosit mai înainte, dar a fost sarita.

Acum sa ne întoarcem la I6, I7 si I8 din Fig. 5.32. Aici vedem ca I6 calculeaza o valoare în R1 care este utilizata de I7. Totusi, vedem si ca valoarea nu este utilizata din nou pentru ca I8 supraînscrie R1.

Nu exista nici un motiv real sa se utilizeze R1 ca loc de pastrare a rezultatului lui I6. Mai rau, R1 este o alegere teribila de registru intermediar, desi perfect rezonabila pentru un compilator sau programator obisnuit cu ideea executiei secventiale fara suprapunre de instructiuni.

În Fig 5.32 introducem o noua tehnica pentru rezolvarea acestei probleme: redenumirea registrelor (register renaming). Unitatea inteligenta de decodificare schimba utilizarea lui R1 în I6 (ciclul 3) si I7 (ciclul 4) într-un registru secret, S1, invizibil programatorului. Acum I6 poate fi lansata concurent cu I5. UCP-urile moderne au adesea zeci de registre secrete folosite pentru redenumirea registrelor. Aceasta tehnica poate elimina adesea dependentele WAR si WAW.

					Registre care sunt citite								Registre care sunt scrise							
C	#	Decodificat	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	-		3	2	1									1	1	1		
3	5	R7=R1*R2	5		3	3	2									1	1	1		1
	6	S1=R0-R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3*R1	-		3	4	2		1							1		1	1	1
	8	S2=R4+R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				6				1								1				
7				4		1	1	1								1				1
8				5				1								1				
9			7																	

C = Ciclu; Iss = Lansata; Ret = Retrasa

Figura 5.32. Functionarea unei UCP superscalar cu lansare în ne-ordine si terminare în ne-ordine.

La I8 utilizam din nou redenumirea registrelor. De aceasta data R1 este redenumit în S2 asa ca adunarea poate fi lansata înainte ca R1 sa fie liber, la sfârsitul ciclului 6. Daca apare necesitatea ca de aceasta data rezultatul sa fie chiar în R1, continutul lui S2 poate fi oricând copiat acolo la timp. Chiar mai bine, toate instructiunile urmatoare care au nevoie de el pot avea sursele lor redenumite la registrul unde este într-adevar memorat. În orice caz, adunare trebuie sa înceapa mai devreme în acest fel.

Pe multe masni reale redenumirea este strâns legata de modul în care sunt organizate registrele. Exista multe registre secrete si o tabela care pune în corespondenta registrele vizibile programatorului în registre secrete. Astfel registrul real utilizat pentru, sa spunem, R0 este localizat urmarind intrare 0 a acestei tabele de corespondenta. În acest fet, nu exista registru real R0, ci numai o legatura între numele R0 si unul dintre registrele secrete. Aceasta legatura se schimba frecvent în timpul exectiei pentru a evita dependentele.

Observati ca în Fig. 5.32 când se citeste de sus în jos coloana a patra, instrutiunile nu a fost lansate în ordine. Nici nu au fost retrase în ordine. Concluzia acestui exemplu este simpla: utilizând executia în ne-ordine si redenumirea registrelor suntem capabili sa marim viteza calculatolui printr-un factor apropiat de doi.

5.2.7. Executie speculativa

În sectiunea precedenta am introdus conceptul de reordonare a instructiunilor pentru îmbunatatirea performatei. Desi nu am mentionat explicit punctul central a fost reordonarea instructiunilor în cadrul unui singur bloc. Este acum momentul sa privim acest aspect mai îndeaproape.

Programele pentru calculatoare se pot sparge în blocuri de baza (basic blocks), fiecare bloc de baza constând într-o secventa liniara de cod cu un punct de intrare în capatul de sus si o iesire în capatul de jos. Un bloc de baza nu contine nici un fet de structuri de control (de exemplu instructiuni if sau while) astfel încât translatarea sa în limbaj masina nu contine nici un fel de ramificatii. Blocurile de baza sunt conectate prin instructiuni de control.

Un program în aceasta forma poate fi reprezentat ca un graf orientat, asa cum este aratat în Fig. 5.33. Aici calculam suma cuburilor întregilor pari si impari, pâna la o anumita limita si acumulam rezultatele lor în evensum si respectiv, oddsum. În cadrul fiecarui bloc de baza, tehnicile de reordonare din sectiunile precedente functioneaza bine.

Necazul este ca majoritatea blocurilor de baza sunt scurte si nu exista suficient paralelism în acestea pentru a-i exploata efectiv. În consecinta pasul urmator este sa se permita reordonarea pentru a intersecta

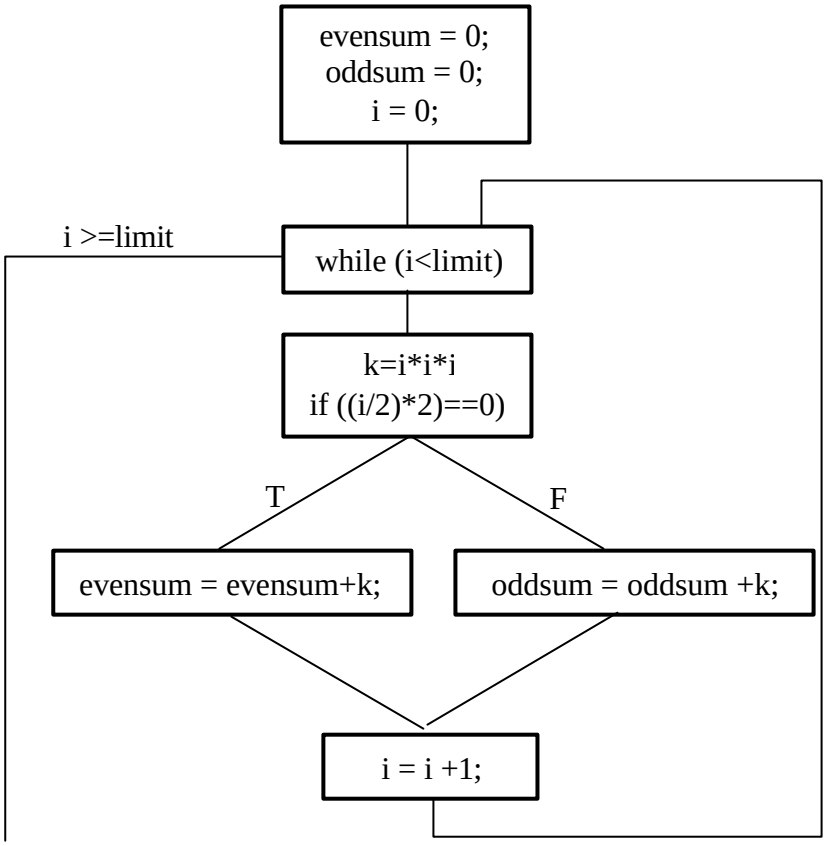
granitele blocurilor de baza în încercarea de a umple toate pozitiile de lansare. Cel mai mare câstig apare când o operatie posibil lenta poate fi mutata mai sus în graf pentru a începe mai devreme. Aceasta at putea sa fie instructiune LOAD, o operatie în virgula mobila sau chiar startul unui lant lung de dependente. Mutarea de cod în sus peste o ramificati se numeste înaltare (hoisting).

```
evensum = 0;
oddsum = 0;
i = 0;

while (i < limit){

k = i * i * i
if ((i/2)*2==0)

evensum = evensum+k
else
oddsum = oddsum + k
i = i + 1
}
```



(a) (b)

Fig. 5.33.(a) Un fragment de program. (b) Graful corespunzator blocului

Sa ne imaginam ca în Fig. 5.33 toate variabilele au fost tinute în registre, exceptând evensum si oddsum (din lipsa de registre). Pare sa existe un sens deci sa se mute instructiunile lor LOAD în vârful buclei, înainte de calcularea lui k, pentru a le începe mai devreme, astfel încât valorile sa fie disponibile când vor fi necesare. Desigur, numai una din ele va fi necesara la fiecare iteratie, asa ca celalalt LOAD (va fi irosit, dar, daca memoria intermediara si memoria principala sunt în banda de asamblare si daca sunt disponibile pozitii de lansare s-ar putea sa merite. Executia codului înainte de a sti daca va fi necesar într-adevar, se numeste executie speculativa (speculative execution). Utilizarea acestei tehnici necesita sprijin din partea compilatorului si a hardware-ului, precum si anumite extensii arhitecturale. În majoritatea cazurilor, reordonarea instructiunilor peste granitele blocurilor de baza este mai presus de capacitatea hardware-ului, deci compilatorul trebuie sa mute instructiunile explicit.

Executia speculativa introduce anumite probleme interesante. Prima ar fi ca, este esential ca una din instructiunile speculative sa nu existe rezultate irevocabile, pentru ca s-ar putea constata mai târziu ca n-ar fi trebuit executate. În Fig. 5.33, se poate citi evensum si oddsum si de asemenea se poate face adunarea imediat ce k este disponibil (chiar inainte de instructiunea if), dar nu este bine sa se memoreze rezultatele în memorie. În secvente de cod mai complicate, o solutie banala de a preveni scrierea registrelor de catre codul speculativ, înainte sa se stie daca acest lucru este dorit, este sa se redenumeasca toate registreje destinatie utilizate de codul

speculativ. În acest fel, sunt modificate numai registrele temporare, asa ca nu este nici o problema daca în ultimul moment codul nu mai este necesar. Daca codul este necesar registrele temporare sunt copiate în adevaratele registre destinatie. Asa cum va puteti imagina, mentinerea tabelei de marcaj pentru pastrarea tuturor acestor informatii nu este simpla, dar se poate face având suficient hardware.

Mai exista însa o alta problema introdusa de codul speculativ, care nu se poate rezolva prin redenumirea registrelor. Ce se întâmpla daca o instructiune executata speculativ cauzeaza o exceptie?

Un exemplu neplacut, dar nu fatal, este o instructiune LOAD care cauzeaza o ratare în memoria intermediara pe o masina cu o dimensiune mare a liniei de memorie intermediara (sa zicem, 256 de octeti) si o memorie cu mult mai lenta decât UCP si memoria intermediara. Daca un LOAD, care este de fapt necesar, opreste functionarea masinii pentru mai multe cicluri pâna când linia de memorie intermediara este încarcata. Nu este alta solutie din moment ce cuvântul este necesar. Totusi blocarea masinii pentru citirea unui cuvânt care se dovedeste a nu fi necesar este contraproductiva. Prea multe astfel de "optimizari" pot face UCP mai lenta decât daca nu le-ar avea deloc. (Daca masina are memorie virtuala, care este discutata în capitolul 6, un LOAD speculativ poate chiar sa genereze o lipsa de pagina, care necesita o operatie cu discul pentru a încarca pagina dorita. Evenimentele false lipsa de pagina pot avea un efect teribil asupra performantei, asa ca este important sa fie evitate.)

O solutie prezenta într-o serie de masini modern este sa se dispuna de o instructiune speciala SPECULATIVE_LOAD care încearca sa citeasca cuvântul din memoria intermediara, dar daca nu este acolo, renunta. Daca valoarea este acolo când este de fapt necesara, se poate folosi, dar daca nu, hardware-ul trebuie sa o aduca imediat. Daca valoarea nu este de fapt necesara, nu se plateste nici o penalizare pentru ratarea în memoria intermediara.

O situatie mult mai rea se poate ilustra cu urmatoarea instructiune:

```
if (x>0) z = y/x;
```

unde x, y si z sunt variabile în virgula mobila . Sa presupunem ca toate variabilele sunt citite în registre în avans si împartirea în virgula mobila (lenta) este înaltata deasupra testului if. Din nefericire, x este 0 si capcana împartire-prin-zero termina programul. Rezultatul net este ca speculatia a cauzat esecul unui program corect. Mai rau decât atât, programatorul a pus cod explicit pentru a preveni acest lucru si acesta s-a întâmplat.

O posibila solutie este sa se dispuna de versiuni speciale de instructiuni care ar putea cauza exceptii. În plus, un bit, numit bit otrava (poison bit), este adunat la fiecare registru. Când o instructiune speculativa speciala esueaza, în loc sa se genereze o capcana se seteaza bitul otrava din registrul rezultat. Daca acest registru este mai târziu accesat de o instructiune obisnuita, atunci se activeaza capcana (asa com ar trebui). Totusi daca rezultatul nu este utilizat niciodata, bitul otrava este sters în cele din urma si nu se întâmpla nimic rau.

5.2.8. Microarhitectura UCP Pentium II

Pentium II admite operanzi si operatii aritmetice pe 32 biti, ca si operatii în virgula mobila pe 64 biti. Admite de asemenea operanzi si operatii pe 8 si 16 biti, care sunt o mostenire de la procesoarele mai vechi din familie. El poate adresa pâna la 64 de GB de memorie si citeste în memorie cuvinte de 64 biti dintr-o data. Un sistem tipic Pentium II este ilustrat în fig. 5.34, care este o reprezentare simplificata a procesorului din fig. 5.27.

Asa cum s-a discutat mai devreme, placheta Pentium II SEC consta din doua circuite integrate: UCP (incluzând memoriile intermediare de nivel 1 separate) si memoria intermediara de nivel 2 unificata. Fig. 5.34 arata componentele primare ale UCP: unitatile Citire/Decodicicare, Repartizare/Executie si Retragere, care lucreaza împreuna ca o banda de asamblare de nivel înalt. Aceste trei unitati comunica printr-un rezervor de instructiuni, un loc pentru pastrarea informatiilor legate de instructiuni partial executate. Informatiile din rezervorul de instructiuni sunt memorate într-o tabela numita ROB (ReOrder Buffer, rom: buffer de reordonare). Pe scurt unitatea de Citire/Decodificare citeste instructiunile si le sparge în micro-operatii pentru memorarea în ROB. Unitatea de Repartizare/Executie preia micro-operatiile din ROB si le executa. Unitatea Retragere încheie executia fiecarei micro-operatii si actualizeaza registrele. Instructiunile sunt introduse în ROB în ordine, pot fi executate în orice ordine, dar sunt retrase din nou în ordine.

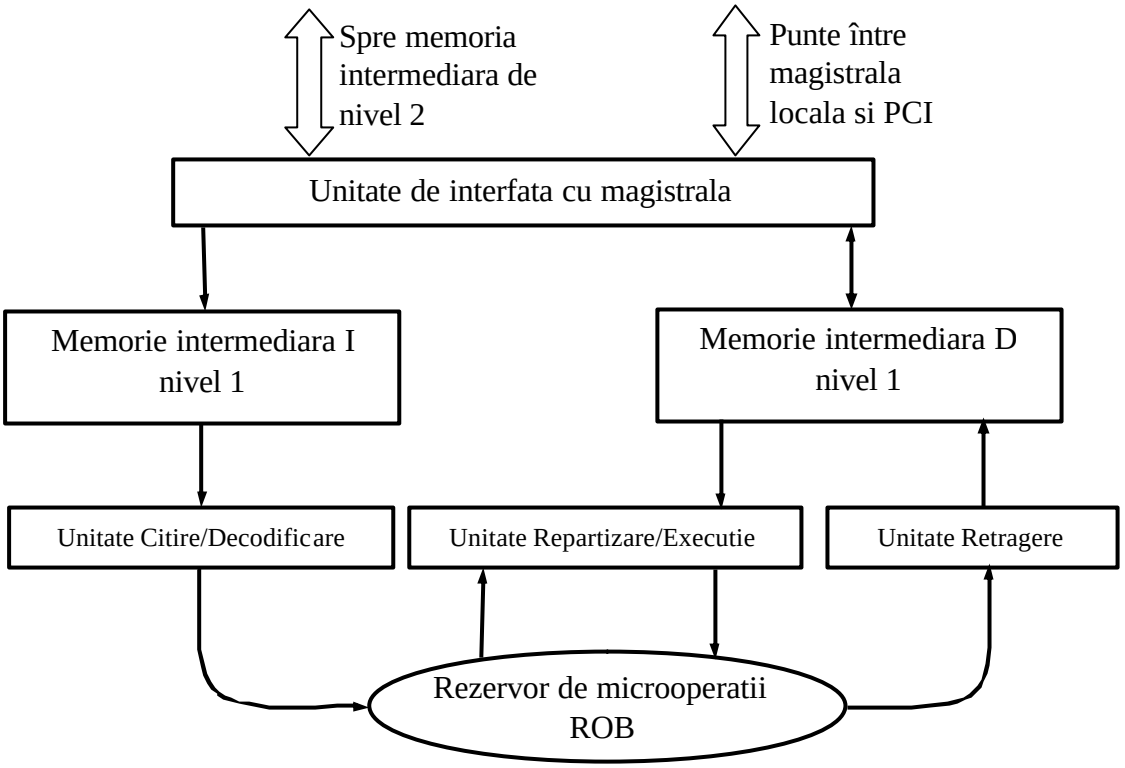


Fig. 5.34. Microarhitectura Pentium II

Unitatea de interfata cu magistrala raspunde de comunicarea cu sistemul de memorie, atât cu memoria intermediara L 2 cât si cu memoria principala. Memoria intermediara L2 nu este conectata la magistrala locala, asa ca unitatea de interfata este responsabila cu citirea datelor din memoria principala prin magistrala locala si încarcarea tuturor memoriilor intermediare. Pentium II utilizeaza protocolul MESI de coerenta a memoriei intermediare, pe care îl vom descrie când vom ajunge la multiprocesoare.

Unitatea Citire/Decodificare

Unitatea Citire/Decodificare este implementata ca banda de asamblare, cu sapte segmente, etichetate de la IFU0 la ROB în Fig. 5.35 (unitatile Repartizare/Executie si Retragere au alte cinci segmente, formând în total o banda de asamblare cu 12 segmente). Instructiunile întra în banda de asamblare în segmentul IFU0, unde sunt încarcate linii întregi de 32 octeti din memoria intermediara I. Ori de câte ori tamponul intern este gol, se copiaza în el o alta linie din memoria intermediara. Registrul NEXT ghideaza procesul de citire.

Deoarece setul de instructiuni Intel, adesea numit IA-32, are instructiuni de lungime variabila, cu mai multe formate, urmatorul segment al benzii de asamblare, IFU1, analizeaza fluxul de octeti pentru a localiza adresa de start a fiecarei instructiuni. Daca este necesar, IFU1 poate cauta în avans pâna la 30 de instructiuni IA-32. Din nefericire, mergând atât de departe înainte, întâlneste patru sau cinci ramificatii conditionate, predictiile nu pot fi corecte pentru fiecare, asa ca valoarea cautarii atât de mult în avans este mica. Segmentul IFU2 aliniaza instructiunile astfel ca urmatorul segment le poate decodifica usor.

Decodificarea începe în segmentul ID0. Decodificarea în Pentium II consta în conversia fiecarei instructiuni IA-2 în una sau mai multe micro-operatii. Instructiunile IA-32 mai simple, ca transferurile registru la registru, se pot converti într-o singura micro-opratie. Altele mai complexe pot necesita pâna la patru micro-operatii. Câteva extrem de complexe necesita chiar mai multe si folosesc ROM-ul secventiatorului de micro-operatii pentru a genera secventa.

Segmentul IDO are trei decodificatoare interne. Doua din ele sunt pentru instructiuni simple; al treilea le gestioneaza pe celelalte. Ceea ce iese din segmentul IDO este o secventa de micro-operatii. Fiecare micro-operatie contine un cod de operatie, doua registre sursa si un registru destinatie.

Micro-operatiile sunt plasate în coada în segmentul ID1. Acest segment mai executa si detectia ramificatiilor. Mai întâi se face o predictie statica, pentru orice eventualitate. Predictia depinde de o serie de factori, dar pentru ramificatiile relative la instructiunea curenta se presupune ca vor fi efectuate cele înapoi, iar cele înainte nu vor fi efectuate. Dupa aceasta urmeaza predictorul dinamic de ramificatii, utilizând un algoritm bazat pe istorie (se va vedea când se va trata memoria cache). În loc sa utilizeze numai doi biti pentru istorie el utilizeaza 4 biti. Trebuie sa fie clar ca pentru o banda de asamblare cu 12 segmente penalizarea pentru o eroare

de predictie este enorma, de aceea se utilizeaza asa de multi biti de istorie. Daca ramificatia nu este în tabela de istorie, se utilizeaza predictia statica.

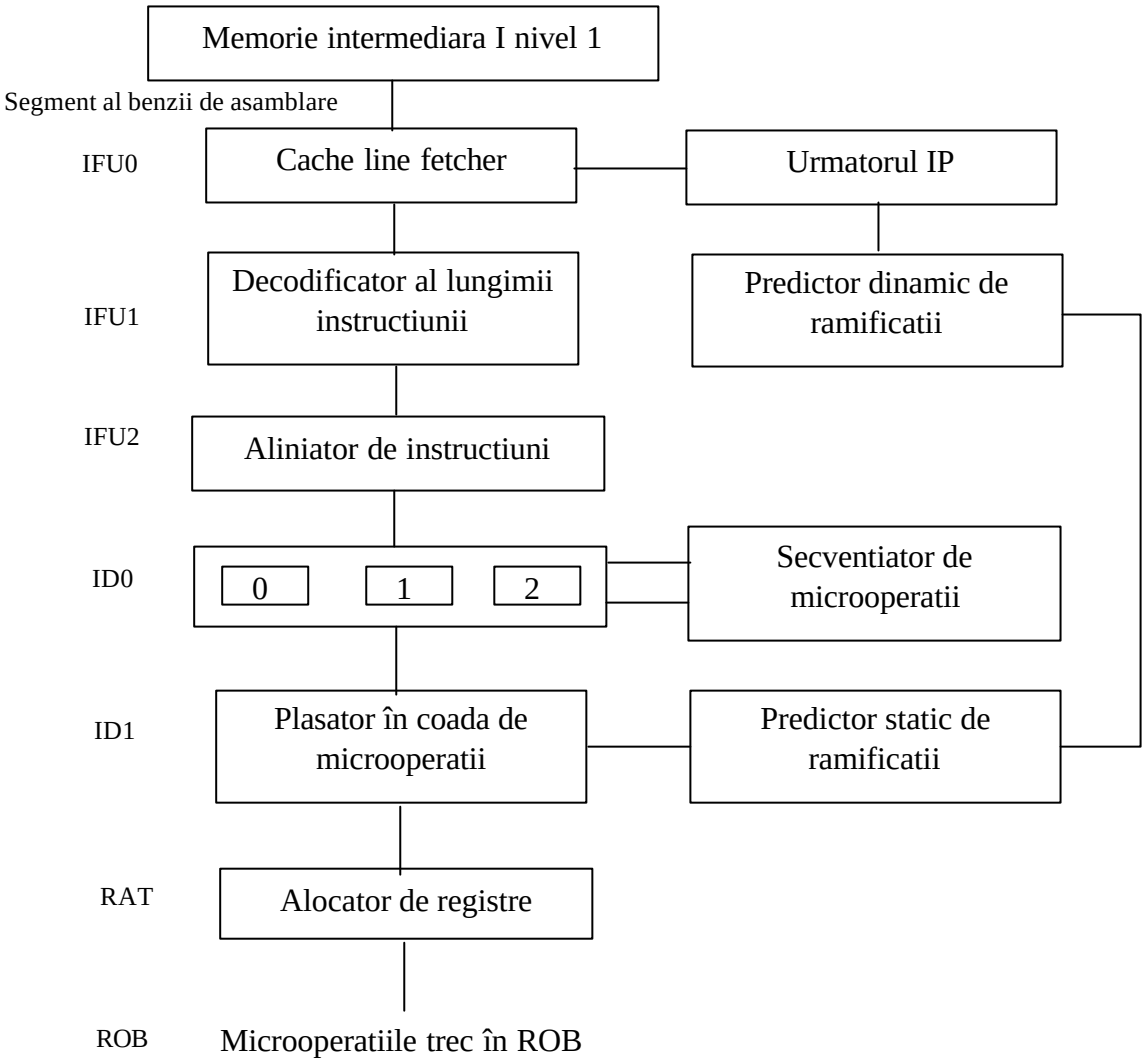


Fig. 5.35. Structura interna a unitatii Citire/Decodificare (simplificata)

Pentru a evita dependentele WAR si WAW, Pentium II permite redenumirea registrelor, asa cum am vazut în Fig. 5.32. Registrele reale numite în instructiunile IA-32 pot fi înlocuite în micro-operatii de oricare din cele 40 de registre temporale interne localizate în ROB. Aceasta redenumire este efectuata în segmentul RAT.

În sfârsit, în ROB sunt depozitate câte trei micro-operatii pe ciclu de ceas. Operanzii sunt de asemenea colectati aici, daca sunt disponibili. Daca operanzii unei micro-operatii si registrul rezultat sunt toti disponibili, iar unitatea de executie este libera, aceasta este candidata sa fie lansata în executie. Altfel, ramâne în ROB pâna când toate resursele sale au fost obtinute.

Unitatea Repartizare/Executie

Trecem acum la unitatea Repartizare/Executie, care este ilustrata în Fig. 5.36. Unitatea de Repartizare/Executie planifica si executa micro-operatii, rezolvând dependentele si conflictele de resurse. Desi numai trei instructiuni ISA pot fi decodificate pe ciclu de ceas (în ID0) pot fi lansate în executie pâna la cinci micro-operatii într-un ciclu, câte una pe fiecare port. Aceasta rata nu poate fi sustinuta pentru ca ea depaseste capacitatea unitatii Retragere.

Micro-operatiile pot fi lansate în orice ordine, dar unitatea Retragere le retrage în ordine. Se utilizeaza o tabela de scor complexa pentru a gestiona micro-operatii în desfasurare, registre si unitati de executie. Când o micro-operatie este eligibila pentru executie ea poate fi începuta, chiar daca altele puse mai devreme în ROB nu

sunt gata. Când mai multe micro-operatii sunt eligibile pentru executie, de catre aceeasi unitate de executie, un algoritm complex o alege pe cea mai buna pentru a fi urmatoarea lansata în executie. De exemplu, executia unei ramificatii este mult mai importanta decât executia unei instructiuni aritmetice, deoarece prima afecteaza fluxul programului.

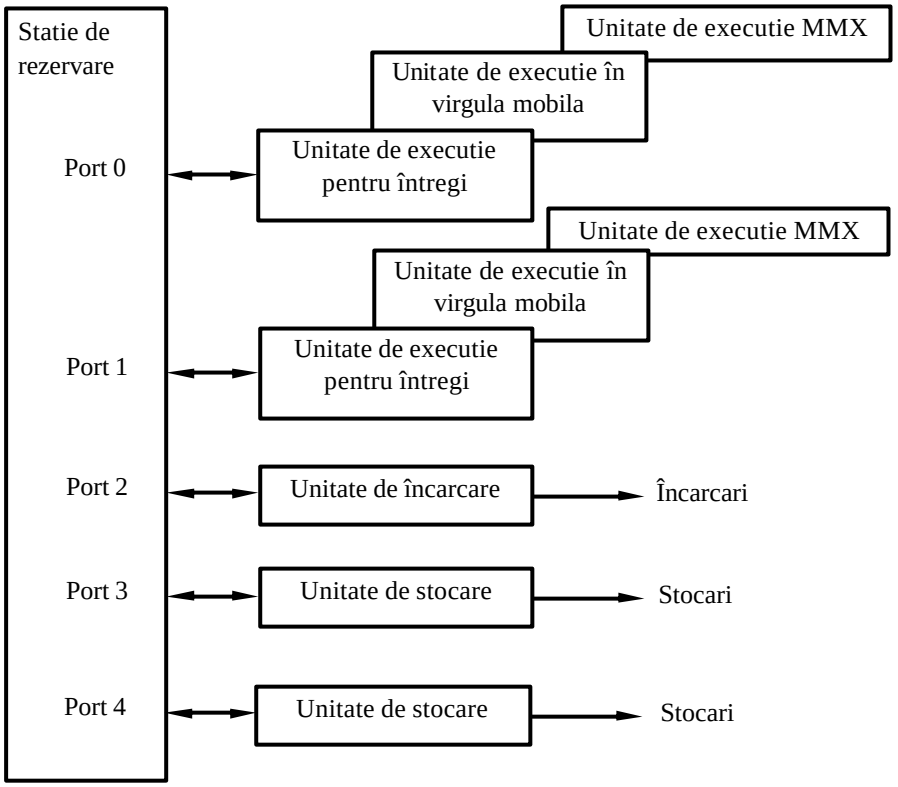


Fig. 5.36. Unitatea de Repartizare/Executie

Unitatea de Repartizare/Executie contine o statie de rezervare si unitati de executie conectate la cinci porturi. Statia de rezervare este o coada cu 20 intrari pentru micro-operatii care au toti operanzii disponibili. Acestea îsi asteapta rândul în Statia de rezervare pâna când unitatea de executie ceruta devine libera.

Exista cinci porturi de la Statia de rezervare la unitatile de executie. Unele unitati de executie partajeaza un singur port, asa cum s-a aratat. Unitatile de încarcare si Memorare initiaza informatiile corespunzatoare pentru operatiile de încarcare si respectiv memorare. Exista doua porturi pentru memorari. Deoarece numai o singura micro-operatie poate fi lansata per port si per ciclu, daca doua micro-operatii în desfasurare trebuie trimise la acelasi port, una din ele trebuie sa astepte.

Unitatea de Retragere

O data ce o micro-operatie a fost executata, aceasta merge înapoi la Statia de Rezervare si apoi în ROB pentru a astepta retragerea. Unitatea de Retragere este responsabila pentru trimiterea rezultatelor la locul adecvat - la registrul adecvat, dar si la alte statii din Unitatea Repartizare/Executie asteptând valori. Unitatea de Citire/Decodificare pastreaza registrele "oficiale", adica acele valori de la instructiuni care s-au terminat. Unitatea de Retragere pastreaza un set de registre în asteptare, adica acele valori care au fost calculate într-o instructiune care nu s-a terminat pentru ca unele instructiuni precedente nu sunt înca terminate.

Pentium II admite integral executia speculativa, asa ca unele instructiuni vor fi executate inutil si nu vor fi retrase. Aici apare capacitatea de executie înapoi. Daca se constata ca o anumita micro-operatie provine dintr-o instructiune IA-32 care nu ar fi trebuit executata, rezultatele sale sunt eliminate. Este sarcina unitatii de Retragere de a gestiona toate acestea. Pot fi retrase numai instructiuni executate "oficial", iar acestea trebuie retrase în ordinea programului, chiar daca ele au fost poate executate în alta ordine.